Spidle: A DSL Approach to Specifying Streaming Applications

Charles Consel[1], Hedi Hamdi[1], Laurent Réveillère[1]
Lenin Singaravelu[2], Haiyan Yu[1*], and Calton Pu[2]

[1] INRIA/LaBRI
ENSEIRB 1, avenue du docteur Albert Schweitzer,
Domaine universitaire - BP 99, F-33402 Talence Cedex, France
{consel,hamdi,reveillere}@labri.fr,
WWW home page: http://compose.labri.fr

[2] College of Computing, Georgia Institute of Technology
801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA
{calton,lenin}@cc.gatech.edu,
WWW home page: http://www.cc.gatech.edu

Abstract. Multimedia stream processing is a rapidly evolving domain which requires much software development and expects high performance. Developing a streaming application often involves low-level programming, critical memory management, and finely tuned scheduling of processing steps.
To address these problems, we present a domain-specific language (DSL) named *Spidle*, for specifying streaming applications. Spidle offers high-level and declarative constructs; compared to general-purpose languages (GPL), it improves robustness by enabling a variety of verifications to be performed.
To assess the expressiveness of Spidle in practice, we have used it to specify a number of standardized and special-purpose streaming applications. These specifications are up to 2 times smaller than equivalent programs written in a GPL such as C.
We have implemented a compiler for Spidle. Preliminary results show that compiled Spidle programs are roughly as efficient as the compiled, equivalent C programs.

1 Introduction

The development of multimedia streaming applications is becoming an increasingly important software activity to account for frequently changing requirements. More and more new formats compete to structure the main media types, creating an explosion in format converters. The need for continuous innovation

* Author's current address: Institute of Computing Technology, Chinese Academy of Sciences. P.O.Box 2704, 100080, Beijing, China. E-mail: yuhaiyan@ict.ac.cn

in the multimedia device industry has shifted an increasing part of stream processing from hardware to software, to shorten the time-to-market [12].

Fortunately, the development of streaming applications relies on well understood libraries of operations for filtering, converting or degrading multimedia streams (*e.g.,* Sox [13]). Furthermore, to account for various application requirements, many implementation variants of common stream operations are often available.

Yet, due to the lack of programming language support, the development of streaming applications tends to be labor-intensive, cumbersome and error-prone: it involves low-level manipulation to cope with bit-level data layout of stream formats, complicated plumbing of components, critical memory management, and meticulous scheduling of processing steps. These difficulties are compounded by the performance critical nature of most streaming applications. As a result, streaming programs are typically manually-optimized for time, and often for space in the case of embedded systems.

This Paper

This paper introduces a domain-specific language (DSL) [1, 2] named Spidle, for developing streaming applications. This language enables high-level and declarative programming of streaming applications without performance loss. Domain-specific verifications are performed on Spidle programs to enhance their robustness.

Domain specific. The design and development of Spidle is based on a thorough analysis of the domain of streaming applications. This analysis has included the study of various specifications of standardized streaming applications [4, 7] as well as typical streaming programs.

High level. Spidle offers high-level constructs and data types that enable programmers to concisely express stream processing. Domain-specific data types and attributes capture dedicated aspects of some values. Domain-specific constructs abstract over common program patterns.

Declarative. A Spidle programmer need only specify the treatment of a given stream; the compiler then maps the specification into an efficient implementation. Information required to trigger domain-specific optimizations is captured in the Spidle program.

Robust. Spidle is safer than a general-purpose language because its syntax and semantics enable domain-specific verifications. In particular, the Spidle compiler checks the consistency of component composition and memory behavior.

The idea of a language dedicated to stream processing has already been discussed in existing literature. Nevertheless, existing approaches are either limited to introducing a language for gluing components of a stream library [19], or geared towards exploiting the features of a specific hardware platform [5, 6].

Contributions

This paper makes the following contributions:

- We have identified common aspects and key concepts used in the development of streaming applications, based on a close examination of various streaming programs as well as specifications of standardized streaming applications.
- We present the definition of Spidle, a high-level and declarative language dedicated to the specification of streaming applications. The language is strongly typed and enables various consistency checks. The resulting degree of robustness of a Spidle program goes beyond what can be achieved with an equivalent program written in a general-purpose language.
- We show that Spidle is highly expressive. It has been used to describe a wide range of streaming applications (see our web site [14]), including traditional ones like a GSM encoder, a GSM decoder, and an MPEG-1 audio encoder as well as special-purpose streaming applications such as Earwax effect, which adjusts CD-audio to headphones [3], and Audio Mixer, which mixes two stereo audio streams into one mono stream [13].
- We demonstrate that Spidle is concise. Our Spidle programs are up to 2 times smaller than equivalent C programs.
- We have implemented a compiler for Spidle programs. The generated code is as efficient as equivalent programs written in C.

Paper Overview

Section 2 presents the difficulties involved in developing a streaming application. Section 3 introduces the Spidle language, focusing on the main language abstractions. Section 4 gives an overview of the compilation process, and lists the main verifications and optimizations performed on a Spidle program. Section 5 assesses the approach. Section 6 presents the related work, and Section 7 concludes the paper and discusses future work.

2 Difficulties in Developing a Streaming Application

In this section, we discuss the issues involved in developing a streaming application and illustrate them with two working examples, namely, GSM encoding and usage of the Sox library. We first briefly introduce these examples.

2.1 Working Examples

GSM transcoding (the process of coding and decoding) enables speech to be transmitted to a digital cellular telecommunication system. The speech signal is compressed before its transmission, thus reducing the size of its digital representation while keeping an acceptable quality of the decoded output. The GSM

coder works on a 13-bit uniform pulse-code modulation (PCM) speech input signal, sampled at 8KHz. The input is processed on a frame-by-frame basis, with a frame duration of 20 ms (160 samples). The full rate encoder [4] presented in this paper transforms a frame of 160 samples into a block of 260 bits, leading to a bit rate of 13 Kbps.

Sox is a library of audio stream processing components. It offers a command line interface that enables an audio file to be converted from one format to another. Various effects and filters can be inserted in the conversion process. Examples include adding an echo, swapping channels, and band pass/reject filters. Additionally, the command line interface enables audio files to be recorded and played.

2.2 The Difficulty of Mapping a Streaming Specification into a Program

A streaming application is often specified informally using a graph-like notation. A node represents a stream filter which transforms particular parts of a stream item. An edge defines the flow of the stream items. Although this notation is convenient at a conceptual level, it can be complex to map such a specification into an implementation. While a specification typically describes some stream tasks as being performed in parallel, an implementation needs to invoke the corresponding components sequentially. This mapping needs to take into account implementation details of the stream tasks involved, such as the possibility of side-effects to a global state. Individual stream tasks require specific data layouts, which entail data conversion. Parts of a stream item may correspond to bit fragments, which must be accessed using low-level bit operators.

Example. A simplified version of the standardized GSM full-rate speech encoding diagram [4] is depicted in Figure 1. The input speech frame is first preprocessed to produce an offset-free signal, which is then subjected to a first order pre-emphasis filter ("Preprocess" in the figure). The 160 samples obtained are then analysed to determine the coefficients for the short term analysis filter (LPC). These parameters are then used to filter the 160 samples. The result is 160 samples of the short term residual signal (STA). For the remaining operations, the speech frame is divided into 4 sub-frames each containing 40 samples of the short term residual signal. Each sub-frame is processed by the subsequent functional elements – we refer to these elements as "Sub-Frame Processing".

Although this is a simplified view of the GSM encoding process, it shows the tangled paths and stages involved in forming the 260-bit encoded block: stream items need to be split, merged and shared across various stages. These intricacies require special care to be mapped into an efficient implementation.

2.3 The Need to Manually Optimize a Streaming Program

The high volume of stream items to process and the stringent real-time constraints to satisfy translate into high-performance expectations when developing

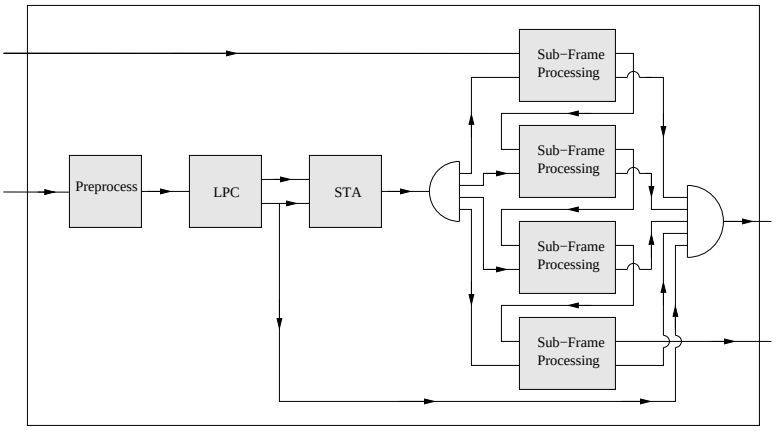

Fig. 1. GSM RPE-LTP Speech Encoding Diagram

a streaming program. As a result, the programmer has to manually perform a number of optimizations, until the performance and resource usage goals are attained. A streaming program not only requires local optimizations, such as loop transformations, but it also relies on global optimizations mostly centered around memory management.

Example. The implementation of GSM full-rate speech encoding, as provided by Jutta and Carsten [8], contains a number of manual optimizations such as code inlining.

2.4 The Need to Manually Optimize Memory Management

Streaming applications typically minimize data copying to reduce the cost of memory management. To apply this strategy, two major aspects need to be taken into account: (1) For efficiency reasons, an implementation of a stream filter often performs side-effects and expects a specific data layout. (2) Most streaming applications not only transform the contents of a stream item but they also change its layout incrementally as it gets processed (*e.g.,* the size of a data fragment expands when it is decompressed).

Two strategies are commonly used to improve the memory usage of a streaming application. One strategy is to schedule stream filters in a particular order depending on their side-effects so as to minimize copying. The other strategy is to allocate memory according to the output data layout, as early as possible in the streaming process, to reduce temporary memory fragments.

Example. The implementation of the GSM encoder is optimized to minimize copying, minimize allocation of temporary buffers and maximize data locality.

For example, consider the Sub-Frame Processing filter shown in Figure 1. The components making up this filter are depicted in Figure 2. The 40-bit residual signal calculated by the long term predictor (LTP) filter is fed to the regular pulse excitation (RPE) filter as a 50-bit signal where the five highest and lowest bits have been padded with zeroes. Memory usage is reduced by propagating the need for a 50-bit buffer backward, to the filter that allocates the incoming 40-bit buffer. This strategy eliminates one memory allocation and one memory copy.

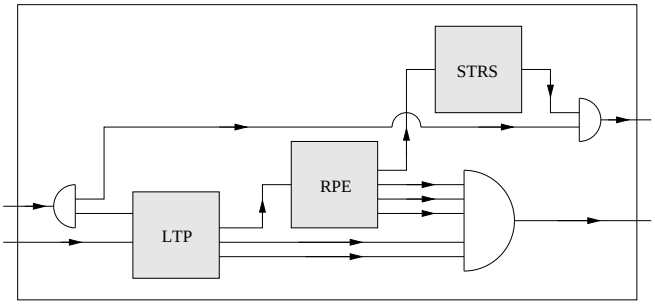

Fig. 2. Sub-Frame Processing

2.5 Error-Prone Re-Use of Stream Filter Implementations

The expected data layout of a stream filter may be incompatible with the one at the current stream stage. This situation requires rewriting a portion of the streaming program, or function wrapping of the stream filter. Although there are many libraries of stream filters, the expertise required for their use often goes beyond the synopsis provided by the library manual. The side-effects of a filter need to be carefully studied to avoid an unexpected data modification at a given streaming stage that corrupts subsequent processing.

Example. Sox filters can be classified based on the type of stream (stereo or mono) they work on. Consider a configuration where the audio stream passes exclusively through stereo filters. Adding a new filter that operates on a mono stream requires a wrapper function that separates the left and right streams before applying the filter and recomposes the stream once the filter is done.

2.6 The Difficulty of Managing Low-Level Code

The need to reduce memory usage implies that very compact data layouts are typically used for stream items. Consequently, accessing individual fields of a stream item often requires low-level bit operations. Such code is known to be hard to develop and to maintain.

Example. Table 1 shows an excerpt of the structure of a 260-bit encoded block generated by the GSM encoder. This description clearly illustrates that compactness of data representation translates into bit-level data layout.

Parameter	Number of bits	Bit n°	Parameter	Number of bits	Bit n°
LARc[0]	6	1 .. 6	xmc[13]	3	110 .. 112
LARc[1]	6	7 .. 12	xmc[14]	3	113 .. 115
LARc[2]	5	13 .. 17	...	3	...
LARc[3]	5	18 .. 22	xmc[25]	3	146 .. 148
LARc[4]	4	23 .. 26	Nc[2]	7	149 .. 155
LARc[5]	4	27 .. 30	bc[2]	2	156 .. 157
LARc[6]	3	31 .. 33	Mc[2]	2	158 .. 159
LARc[7]	3	34 .. 36	xmaxc[2]	6	160 .. 165
Nc[0]	7	37 .. 43	xmc[26]	3	166 .. 168
bc[0]	2	44 .. 45	xmc[27]	3	169 .. 171
Mc[0]	2	46 .. 47	...	3	...
xmaxc[0]	6	48 .. 53	xmc[38]	3	202 .. 204
xmc[0]	3	54 .. 56	Nc[3]	7	205 .. 211
xmc[1]	3	57 .. 59	bc[3]	2	212 .. 213
...	3	...	Mc[3]	2	214 .. 215
xmc[12]	3	90 .. 92	xmaxc[3]	6	216 .. 221
Nc[1]	7	93 .. 99	xmc[39]	3	222 .. 224
bc[1]	2	100 .. 101	xmc[40]	3	225 .. 227
Mc[1]	2	102 .. 103	...	3	...
xmaxc[1]	6	104 .. 109	xmc[51]	3	258 .. 260

Table 1. Data Layout of the Encoded Blocks of the GSM Encoder

3 The Spidle Language

Based on the domain analysis of stream processing, we have identified the following key requirements for a language dedicated to this domain. The language should be flow-based to describe the paths through which stream items are propagated and processed by stream tasks; it should include stream-specific declarations to enable dedicated verifications and optimizations to be performed; it should be module-based to enable a streaming application to be decomposed into manageable components; it should include an interface language to enable disciplined re-use of existing stream filter libraries.

3.1 An Overview of Spidle

A Spidle program essentially defines a network of *stream tasks*. *Flow declarations* specify how stream items flow within stream tasks (nodes) and across stream tasks (edges), as well as the types of these stream items.

A stream task can either be a *connector* or a *filter*. Connectors represent common patterns of value propagation. Filters correspond to transducers; they can either be *primitive* or *compound*. A primitive filter refers to an operation implemented in some other programming language. This facility enables existing filter libraries to be re-used. A compound filter is defined as a composition of stream filters and connectors. This composition is achieved by *mapping* the output stream of a task to the input stream of another task.

Let us now present the abstractions offered by Spidle in detail.

3.2 Flow Declarations

Two abstractions address the flow aspects of a streaming application: a *stream* specifies the flow aspects at the task level; a *mapping* specifies how stream items flow across tasks.

Streams. A stream task declares streams using the type constructor `stream`. A stream declaration defines what type of items flow in a stream and their direction. The first aspect is addressed by an appropriate type language. The second aspect defines how items flow. A stream task can declare a stream to be an input stream, to be an output stream, or both. An input-only stream contains values that flow in, but not out, of the stream task. An output-only stream describes values that are created in the stream task. An input-output stream contains values that are propagated through a stream task.

An example of a stream is displayed below.

```
stream inout int16[40] e;
```

This declaration is extracted from the Spidle definition of the filter `RPE_Encoding` of the GSM encoder. It specifies that values of type `int16[40]` flow both in and out of the filter `RPE_Encoding`.

The stream declarations of a stream task are grouped in a clause named `interface`, as illustrated later in stream task examples.

Mappings. A mapping defines how stream items flow from one stream task to another. Mapping declarations of a stream task are grouped into a clause called `map`. A mapping can either be (1) one-one, (2) one-many or (3) many-one. The first kind is the most common; it connects the output of one stream task to the input of another one. An example of such a mapping is displayed below. It specifies that the value of stream `so` is obtained by padding 5 zero-bits on both sides of stream `si`.

```
map {
   {0,0,0,0,0} # si # {0,0,0,0,0} -> so;
}
```

This `map` clause consists of a single mapping declaration. The left-hand side of the "`->`" sign defines the source stream of the mapping; the right-hand side names the destination stream. As shown in the example, the source stream is represented by a *stream expression*, that is, an expression that constructs a stream value by applying the concatenation operator "`#`" to constants and stream variables.

The one-many mapping is required when the processing of one item produces many items, which are then processed sequentially. This situation is illustrated by the MPEG-1 audio encoder shown in Figure 3. In this encoder, a buffer of PCM samples is split into 24 blocks, each of which is processed by the `SubbandBlock` filter.

A many-one mapping is used when a stream task needs a collection of items before performing an operation. This situation is again exemplified by the MPEG-1 audio encoder (see Figure 3) where the `ScaleFactorCalculator` filter expects to receive all the samples produced by the `SubbandBlock` filter as a single input stream item.

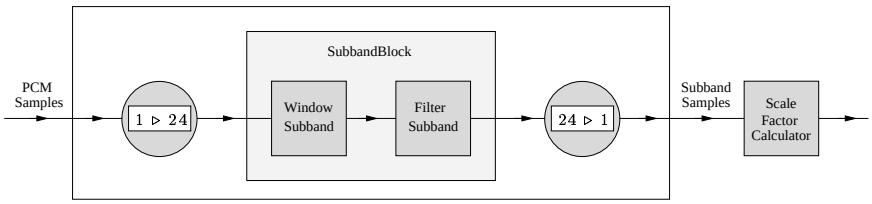

Fig. 3. MPEG Subband Filter for Stereo Streams

We have not found any use for a many-many mapping, except in cases that can be re-expressed using a one-one mapping.

3.3 Stream Tasks

A stream task can either be a connector or a filter. The difference between these two kinds of tasks is that a connector propagates stream items in a fixed way and is guaranteed not to modify their value, beyond what a stream expression enables. These restrictions do not apply to filters.

Spidle offers a type constructor for connectors and one for filters. These type constructors enable the programmer to define a task type. Instances of a new task type can then be created throughout a Spidle program. To improve re-use, a task type can be instantiated with respect to both compile-time and run-time arguments. An example of such instantiation is presented in the filter section below.

Connectors. There are two kinds of connectors: mergers and splitters. A merger fuses independent input streams into a single output stream. A splitter performs

the opposite operation. Because a connector can only be used to link stream tasks, a connector declaration only contains an `interface` clause and a `map` clause. An example of a declaration of a merger from the GSM specification is given below.

```
merger Frame_Merger {
  interface {
    stream in  bit[7] nc;
    stream in  bit[2] bc;
    stream in  bit[2] Mc;
    stream in  bit[13][3] xMc;
    stream in  bit[6] xmaxc;
    stream out bit[56] bits;
  }
  map {
    nc # bc # Mc # xmaxc # xMc -> bits;
  }
}
```

This connector merges input streams `nc`, `bc`, `Mc`, `xMc` and `xmaxc` into the single output stream `bits`.

Filters. A filter can either be compound, when it combines a set of stream tasks, or primitive, when it refers to an operation implemented in some foreign programming language.

Compound filters. A compound filter defines a combination of other stream tasks. As a result, besides the `interface` and `map` clauses, a compound filter consists of an `import` clause referring to the Spidle files defining the needed stream tasks. Since a compound filter imports task types, it also needs an `instantiate` clause to define task instances with respect to a specific context. An example of a declaration of a compound filter from the MPEG-1 audio encoder specification follows.

```
filter SubbandBlock(int stereo) {
   interface {
      stream in  int16[2][384] buffer;
      stream out float64[32]   sample;
   }
   instantiate {
      WindowSubband(stereo) ws;
      FilterSubband fs;
   }
   map {
      buffer -> ws.buffer;
      ws.Z -> fs.Z;
      fs.sb_sample -> sample;
   }
}
```

This filter has a formal parameter, `stereo`, which is given a value at run time, when the filter is instantiated. This value is also used to instantiate the filter `WindowSubband`, as shown in the `instantiate` clause of `SubbandBlock`.

Primitive filters. A primitive filter enables existing library code to be imported into Spidle. Like the compound filter, a primitive filter includes an `import` clause, but this clause refers to files written in some other programming language. Both functions and types can be imported, thus allowing Spidle to propagate foreign values from one primitive filter to another one.

Because a primitive filter provides an interface to a foreign function, it does not include an `instantiate` clause. Instead, it consists of a `run` clause that invokes the foreign function.

The foreign function invoked in a primitive filter may also modify the contents of the buffer attached to an input-only stream using, for example, previously read locations as temporary storage. Spidle requires that the declaration of each input-only stream that is passed to the foreign function specify the effects of this function on the stream items. A stream can be declared to be read by the foreign function (default behavior) or both read and written to by the foreign function. This critical information is later used to optimize memory management.

Let us examine an example of a primitive filter from the GSM specification.

```
filter Weighting {
  interface {
    stream in  bit[50][16] e;
    stream out bit[40][16] x;
  }
  import {
    func Weighting_filter from "rpe.c";
  }
  run {
    Weighting_filter (e, x);
  }
}
```

This filter only reads input stream `e`, and writes output stream `x`. These streams are passed to the foreign function `Weighting_filter` defined in file `rpe.c`. Here, the C programming language is assumed to be the foreign language used in the `run` clause.

3.4 Advanced Features

A network of stream tasks may contain loopbacks (cycles) when a path connects the output of one stream task to one of its inputs. Such a network has special semantics since some items are unavailable as inputs when processing begins.

Spidle offers a built-in task type named *delay* for introducing loopbacks in a network. This specific task simply propagates items from its input stream to its output stream. Such a task type requires at least one compile-time argument at the time of instantiation. This parameter enables the programmer to define

how many initial items have to be produced on the output stream before looking for items on the input stream. In addition, a delay task can also be instantiated with respect to appropriate values of the initial items.

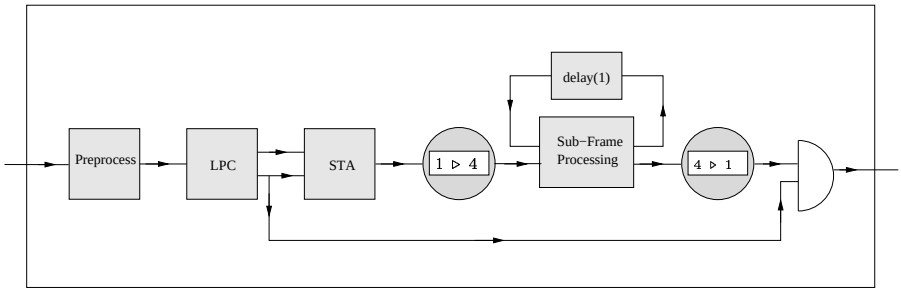

Fig. 4. GSM Speech Encoding Diagram with Loopback

The GSM full-rate speech encoding diagram depicted in Figure 4 illustrates the use of the built-in task delay. In this modified version of the GSM encoding diagram shown in Figure 1, the 160 samples of the short term residual signal are first split into 4 sub-frames before being processed sequentially by the Sub-Frame Processing filter. One of the output streams of this filter is connected to one of its input streams through a delay filter. This task has been instantiated with the value 1 to indicate that it produces a value with a one-step delay. Since no initial value has been defined for this delay task, the first item provided to the Sub-Frame Processing filter is 0. Following items are obtained by looking at the output stream of previous iterations. When all the required items are available, they are merged to form the 260-bit encoded block.

4 Compilation

We now present a preliminary design of the Spidle compilation process. This process consists of the following steps. First, dependencies between stream tasks are collected and represented as a graph. Second, this graph is annotated with effects and memory management information. Third, the resulting graph is used to schedule stream tasks. Fourth, the memory layout of stream items at each stage of the streaming process is computed. Lastly, code is generated.

Dependency graph. Given a Spidle program, the compiler computes the transitive closure of the referred stream tasks by examining the `instantiate` and `import` clauses. The resulting information is represented as a dependency graph.

Effects. This pass determines the effects of each compound filter based on the effects of the primitive filters it references.

Memory management. This analysis annotates the graph of stream tasks with information describing the lifetime of memory fragments. This information is computed using the stream declarations. Specifically, for an output-only stream, memory needs to be allocated to store the value of a stream item. Conversely, for an input-only stream, the memory used for the stream item is not needed beyond the execution of the stream task.

Task scheduling. Most Spidle programs have more than one possible schedule for their stream tasks. In our current design, the scheduling strategy focuses on the choice points represented by splitters. Spidle chooses a schedule that minimizes memory copies, using effect information.

Memory layout. A streaming application often transforms values from one format to another. The transformation is carried out incrementally as the item gets propagated through the various steps of the stream process. This situation introduces temporary memory fragments. To remedy this potential source of inefficiency, our compiler attempts to allocate space eagerly. That is, when the size of an item grows as it gets processed, its final size is used for the initial allocation. Of course this optimization cannot always be applied; for example, when non-contiguous data from input streams are processed, replacing the allocated buffers of input streams by the final buffer may not be allowed.

Currently, Spidle has only been interfaced with C and C++ languages. These are the programming languages used by most stream libraries, mainly for efficiency reasons.

4.1 Verifications

Because of its domain-specific nature, the Spidle compiler can perform a number of verifications that are beyond the reach of a compiler for a general-purpose language. These verifications focus on the composition of stream tasks, the propagation of stream items, and the usage of foreign libraries.

Composition of stream tasks. Stream declarations are checked to ensure that types and directions of stream items are compatible when stream tasks are combined. Inconsistent combinations of effects and directions are detected.

Propagation of stream items. Mappings are inspected to find unconnected streams and input streams connected to more than one output stream. Also, omitted or double definitions can be detected. For example, one and only one mapping declaration must specify how the value of a bit of an output stream is obtained.

Usage of foreign libraries. External function declarations are analyzed to ensure that the types of the actual parameters they accept are equivalent to types specified in the stream declarations that use them.

4.2 Optimizations

In fact, most streaming applications are targeted for use in embedded systems. Such systems usually have moderately powered processors, minimal amounts of memory and limited battery. To stay within these limitations, streaming programs are typically optimized manually.

The domain-specific constructs of Spidle open up opportunities for various optimizations that are not possible when using a general-purpose language. The goals of these optimizations are to reduce the number of memory copy operations, to reduce memory usage, to improve data and code locality, and to reduce the size of the resulting code. The order of this list reflects roughly the order of the significance of these optimizations, although it may vary considerably in certain scenarios depending on their specific requirements.

For example, in embedded systems, it is often more desirable to keep memory utilization below a certain threshold than to minimize it as much as possible.

In contrast with manual optimizations dedicated to a given architecture and a specific streaming application, the Spidle compiler automatically performs global optimizations that are not possible locally. Because the engine is parametrized, optimizations are retargetable without any additional effort to a new system that has different resource constraints, such as memory usage and cache sizes.

5 Assessment

The assessment our language is a crucial step of the DSL approach. In our experience [16–18, 11, 10], a DSL should be assessed with respect to three pragmatic criteria: expressiveness, conciseness, and performance.

Expressiveness. Assessing the expressiveness of a DSL in practice requires to use it for a variety of non-trivial applications. We have used Spidle to express a GSM encoder, a GSM decoder and an MPEG-1 audio encoder. These applications must satisfy industrial-strength standards, and are commonly mentioned as reference cases to assess work in the domain of stream processing (*e.g.,* [19, 15]). We have also specified other, more dedicated, streaming applications referenced in various libraries, toolkits and middleware for stream processing.

Conciseness. Because Spidle offers domain specific abstractions and constructions, it enables the programmer to concisely define a streaming application. We found that Spidle programs are up to 2 times smaller than equivalent version written in C.

Performance. Our performance measurements of the compiled code of Spidle programs show that, at worst, there is a negligeable loss of performance (around 4%) compared with the equivalent C-compiled code written by an outside expert. These results are preliminary and should improve as our compiler gets further developed.

6 Related Work

StreamIt [19] is certainly the work most related to Spidle. It is a Java extension that provides the programmer with constructs to assemble stream components. While StreamIt and Spidle share the same goals, their approaches vary considerably. StreamIt is essentially a GPL that offers extensions corresponding to common program patterns and an interface to library components. Because it is a superset of Java, performance of compiled StreamIt code is as good as what existing Java compilers produce, which is currently much slower than compiled equivalent C code. Indeed, Java has some intrinsic overhead (*e.g.,* memory management and object layout) that may not be easy to work around. Lastly, because a StreamIt program is intertwined with Java code, verification is very local to the domain-specific constructs, unlike what can be done in Spidle.

The Infopipes system [9] is a middleware abstraction for describing information flow in the case of distributed streaming applications. The goal of Infopipes is to expose the underlying communication layer to the application so that it can adapt dynamically to changing network or processing conditions. The Infopipes system offers distributed versions of our splitters, mergers and filters. In contrast, Spidle is limited to local stream processing, and focuses on performance and verification.

Stream-C [5] and Sassy [6] are two stream-oriented languages used to describe hardware circuits. The aim of these languages is to represent circuits with higher level abstractions like processes, streams and signals with the emphasis on reducing the clock rate and the board area occupied by the generated circuit. Spidle operates in a different domain. While it might be possible to write a compiler for Stream-C or Sassy dedicated to streaming applications, their constructs are not well suited to this domain.

7 Conclusions and Future Work

Stream processing is a rapidly evolving field which requires much software development with high-performance expectations. To address these requirements, we have developed a domain-specific, high-level and declarative language named Spidle, for specifying streaming applications.

We have used Spidle to write a variety of industry-standardized streaming applications as well as special-purpose ones. These specifications have experimentally validated the expressiveness of our language. Spidle programs were up to 2 times smaller than equivalent programs written in C.

We have implemented a compiler for Spidle. Preliminary experiments show that compiled Spidle programs have performance that is roughly comparable to compiled equivalent C programs.

Our implementation of the Spidle compiler is preliminary, and there is a number of optimizations that need to be explored. In particular, we plan to optimize locality by taking into account processor features, such as data cache and instruction cache, when determining a scheduling for stream tasks. We also

want to study the performance impact of buffering input stream items before firing the stream process. These ideas are examples of highly domain-specific optimizations that can be enabled by the presence of more explicit information at the language level, and can be factorized into a compiler.

Another track of research aims to go beyond local stream processing to tackle the distributed case. To do so, we are studying ways to integrate Spidle into a middleware for distributed streaming. In particular, we are working on Infopipes [9], partly developed by one of the authors.

Finally, we are working on a graphical representation for Spidle. A visual version of Spidle seems quite a natural step to take considering that the graph-like notation is commonly used in the field. Toward this end, we first plan to build a tool capable of visualizing Spidle programs as a graph of stream tasks.

Acknowledgment.

We thank Julia Lawall, Anne-Françoise Le Meur and the other members of the Compose group for helpful comments and discussions on earlier versions of this paper. We also thank the anonymous reviewers for their valuable inputs.

This research has been partially funded by Conseil Régional d'Aquitaine, DARPA/IXO (PCES program), National Science Foundation grants numbered CCR-9988452, ITR-0121643, ITR-0219902, and 0208953, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment.

References

1. C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the* 10^{th} *International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in Lecture Notes in Computer Science, pages 170–194, Pisa, Italy, September 1998.
2. A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.
3. Earwax effect. http://www.geocities.com/beinges/works.htm.
4. European Telecommunications Standards Institute, 650, route des Lucioles F-06921 Sophia-Antipolis Cedex – France. *GSM full speech transcoding 06.10*, Nov 2000. REN/SMG-110610Q8R1.
5. M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–59, Apr 2000.
6. J. P. Hammes, B. A. Draper, and A. P. Willem Boehm. Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems. *Lecture Notes in Computer Science*, 1542:83–97, 1999.
7. International Organisation for Standardisation, 1, rue de Varembé, Case postale 56 CH-1211 Geneva 20, Switzerland. *Moving Picture Experts Group (MPEG-1 audio) Specifications*, 1993. ISO/IEC 11172-3:1993.
8. D. Jutta and B. Carsten. C implementation of GSM 06.10 RPELTP coder and decoder. http://kbs.cs.tu-berlin.de/ jutta/toast.html, Nov 1994.

9. R. et al. Koster. Infopipes for composing distributed information flows. In *Proceedings of the ACM Multimedia Workshop on Multimedia Middleware*, Oct 2001.
10. F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30, San Diego, California, October 2000.
11. L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A DSL approach to improve productivity and safety in device drivers development. In *Proceedings of the* 15^{th} *IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 101–109, Grenoble, France, September 2000. IEEE Computer Society Press.
12. L. Rizzo. On the feasibility of software FEC. Technical Report LR-970131, Dip. di Ingegneria dell'Informzione, Università di Pisa, Jan 1997.
13. Sox sound exchange. http://www.spies.com/Sox.
14. Spidle home page. http://compose.labri.fr/prototypes/spidle.
15. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
16. S. Thibault and C. Consel. A framework of application generator design. In M. Harandi, editor, *Proceedings of the Symposium on Software Reusability*, pages 131–135, Boston, Massachusetts, USA, May 1997. Software Engineering Notes, 22(3).
17. S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.
18. S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
19. W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 179–196. Springer-Verlag, 2002.